Atty Docket No.: IDF-2476
4000-14300

Express Mail Label No. EL 989053219 US

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE
APPLICATION FOR UNITED STATES LETTERS PATENT

SOFTWARE BUILD TOOL

By:

Bobby B. Brown
4052 North 107th Terrace
Kansas City, KS 66109
Citizenship: USA

Shawn M. Hudson
910 West Elizabeth Street
Olathe, KS 66061
Citizenship: USA

John J. Wright
8042 Mohawk, Apt. 201
Prairie Village, KS 66208
Citizenship: USA

SOFTWARE BUILD TOOL

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] None.

STATEMENT REGARDING FEDERALLY SPONSORED RESEARCH OR DEVELOPMENT

[0002] Not applicable.

REFERENCE TO A MICROFICHE APPENDIX

[0003] Not applicable.

FIELD OF THE INVENTION

[0004] The present invention is directed to computer software, and more particularly, but not by way of limitation, to a system and method for building computer software.

BACKGROUND OF THE INVENTION

[0005] Building a modern software application often is a complicated process. The application or software may comprise multiple executable tasks, modules, or components as well as various data and/or configuration files. Parts of the software may need to be created from computer program source files by compiling the computer programs into object code or into byte code and then linking the object code into an executable (byte code may be run directly by an interpreter and need not be linked to be made executable). Compiling source code may need to take place in a particular order to resolve dependencies between the source code files. In addition to compiling source code, the building process may involve validating either the presence or contents of configuration files and also may involve processing other kinds of files or data. The end products of software build activities may be stored in archive files, for example, in Java archive (JAR)

files. The number of sources and other files that are needed to build an application may be quite large, which adds to the complexity of building the application.

[0006] Building software may comprise both manual and automated steps. Typically, an automated build process is less time consuming and less error prone than a manual build process. Scripts are often employed to automate parts of the build process. A script is a series of commands that may be directly executed by a computer which understands the script language without the series of commands needing first to be compiled and or linked.

[0007] A code control system (CCS) is commonly associated with a software build system. The CCS provides mechanisms for controlling changes to the various files and sources which are employed to build the software as well as for controlling the resultant products of the build process. One of these mechanisms may include a locked check-out which prevents two developers from corrupting each other's work. Without a locked check-out mechanism, for example, if two developers check-out copies of the same file, each developer changes their checked out copy of the same file, each developer checks in their edited copy of the file, the last copy to be checked in writes over the edits of the first copy to be checked in, destroying those edits. With a locked check-out mechanism, only one developer is permitted to check-out a file for writing at a time. Others may check-out readable copies but not editable copies of the locked file. Another mechanism may enforce a policy that files may not be checked-out for writing without providing a reference to an active bug report or authorized software change request. Another mechanism may enforce a policy that files may not be checked-in without providing a reference to results of testing the changed software. A CCS may be purchased or leased as an off-the-shelf

software product, for example Merant's PVCS version management system or IBM's Rational ClearCase software configuration management system.

[0008] The process of building software or building an application is sometimes referred to as conducting a software build, making a software build, or making a build. The result of the process of building software is sometimes referred to as a software build or a build.

SUMMARY OF THE INVENTION

[0009] The present embodiment provides a system for managing software builds. The system comprises a code control system operable to maintain a code version and a information associated with the code version, a parser module in communication with the code control system, the parser module operable to parse the information associated with the code version and create a change report, and a compiler module in communication with the code control system and operable to compile the code version into an object version based on the change report.

[0010] In one embodiment a method of managing software builds is provided comprising changing, by a developer, source code of a software archive, requesting, in a source archive system maintaining the software archive, a build of the software archive including the source code, the request including a build request template, generating a change matrix based on the build request template, notifying an approver of the software build request, notifying the developer when the software build request is denied by the approver, and rebuilding the software archive based upon the change matrix when the software build request is approved by the approver.

[0011] In one embodiment a method for building a software version is provided comprising storing a revised code version and a description of the revisions in a code

control system, generating a change report based on the description of the revisions to the revised code version, authorizing a build of a software version including the revised code version, and building the software version based on the change report.

[0012] These and other features and advantages will be more clearly understood from the following detailed description taken in conjunction with the accompanying drawings and claims.

BRIEF DESCRIPTION OF THE DRAWINGS

[0013] For a more complete understanding of the present disclosure and the advantages thereof, reference is now made to the following brief description, taken in connection with the accompanying drawings and detailed description, wherein like reference numerals represent like parts.

[0014] Figure 1 is a block diagram of the software build tool system according to one embodiment.

[0015] Figure 2 is a block diagram of the software build tool system according to another embodiment.

[0016] Figure 3 is a flow diagram for a method of building a software application.

[0017] Figure 4 is a flow diagram of a detailed portion of the method of building a software application.

[0018] Figure 5 illustrates an exemplary general purpose computer system suitable for implementing the several embodiments of the software build tool.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0019] It should be understood at the outset that although an exemplary implementation of one embodiment of the present disclosure is illustrated below, the present system may be implemented using any number of techniques, whether currently known or in existence. The present disclosure should in no way be limited to the exemplary implementations, drawings, and techniques illustrated below, including the exemplary design and implementation illustrated and described herein.

[0020] The software build tool of the present disclosure combines a Java-based website interface and Jakarta ANT scripts working in coordination with the Merant PVCS version management system to manage the creation of Workflow Broker build java archives (JARs). One objective for the software build tool is to provide a unified tool for introducing changes of JARs, for approving changes to JARs, and for building JARs.

[0021] Turning now to Figure 1, a block diagram of a software build tool system 10 is depicted. A code control system (CCS) 12 stores the code sources, various files, and other input which are processed to build a software product or application. The code sources may include files in various programming languages, for example, Java and C++. The various files may include, for example, configuration text files, interface definition language (IDL) files, and data type definition (DTD) documents. An IDL file contains a language independent definition of how two modules or systems communicate with each other. When two systems communicate by passing text files back and forth between each other, for example by passing extensible markup language (XML) text files, a BusinessWare application must be informed of what these text files contain. BusinessWare applications obtain this information through examining metadata contained

in metadata classes. The DTD documents may be compiled to produce the needed metadata classes, as described further hereinafter.

[0022] The CCS 12 also stores the intermediate and end products of the software build process including object files, compiled byte code files, executable files, and Workflow Broker build JAR files. The Workflow Broker may be a component or application of an commercial off-the-shelf (COTS) application such as Vitria's Business Ware, which has been customized or tailored for these purposes. The CCS 12 also stores files associated with the mechanics of the build process itself including script files and a change report 22. The CCS 12 may be an off-the-shelf software application, for example the Merant PVCS version management system. The CCS 12 may be associated with storage 14 where source files, configuration files, object code files, linked files, and other build product data are physically stored. In one embodiment, the CCS 12 provides the preferred method to access the storage 14.

[0023] The CCS 12 may provide interfaces which users and administrators may use to interact with and use the CCS 12, thereby accessing and interacting with the storage 14. A user web client 16 interface and an administrator web client 18 interface are depicted in Figure 1, but in some embodiments there may be other interfaces. The user web client 16 may check-out files, modify checked-out files, and check-in files. Typically software developers would employ the user web client 16 to develop the software.

[0024] The CCS 12 may enforce a policy requiring the user web client 16 to provide text comments in conformance with a standard comment form when checking-in files. The comments describe the changes to the file, the version of the file being checked-in, perhaps a reference to test results, and other information. The standardization of the form

of the file check-in comment makes the comments parseable by scripts. An example standard comment template is the following:

brief.description=a change

ticket.num=I0007572767

screenshots.update=yes

states.added=

states.removed=

transitions.added=

transitions.removed=

events.added=

events.removed=

fields.added=

fields.removed=

The states, events, transitions, and fields which are associated with the above example standard comment template refer to constructs employed in a software analysis model. For example, a state-transition model may represent one aspect of a software application as a plurality of static states, a set of allowed transitions among the states, and a set of events which may trigger specific transitions from a first state to a second state. Fields may refer to components of an object.

[0025] When a software build is required, for example, when a software module is changed, an event is generated which causes a parser module 20 to parse the standardized comments associated with the designated changed files and generates from these comments a change report 22, also referred to as a change matrix, which

consolidates, for example in one document or file, the information necessary to define or specify the software build to be performed. The change report 22 may contain entries for multiple changed files, wherein one entry in the change report is associated with each changed file. In some embodiments, the Java properties format may be used to parse comments. In one embodiment, the change report 22 may be captured in a Microsoft Excel spreadsheet document, but the software build tool system 10 is not limited to employing a Microsoft Excel spreadsheet document to represent the change report 22. The change report 22 is stored in the CCS 12. The event which triggers the parser module 20 to generate the change report 22 may take different forms in different embodiments. For example, in a PVCS version management system the event may be the developer changing the status of an archive from SUBMIT status to CHANGED status, which is also known as promoting from SUBMIT status to CHANGED status. In other embodiments, the generation of the change report 22 triggers an email to be sent to build administrators that states that a build is waiting for their attention.

[0026] The change report 22 may be authorized by a build administrator using the administrator web client 18 interface. A build administrator examines the change report 22, verifies that software change policies have been adhered to, and approves or authorizes the software build (or disapproves the software build). The software change policies, for example, may require that testing be performed and test results be referenced in the change report 22. The test results reference might be introduced into the change report 22, for example, via a test results field in the standardized comment. When the change report 22 is approved or authorized, an event is generated which causes the software build to start. The event which triggers the software build may take different forms in different

embodiments. For example, in the PVCS version management system the event may be the administrator changing the status of the archive, also referred to as promoting the archive status, from CHANGED to APPROVED status.

[0027] The software build process involves a compiler module 24 compiling source code into object code. The compiler module 24 may comprise one or more scripts which invoke one or more off-the-shelf compiler programs. The compiler module 24 is in communication with the CCS 12 and has access thereby to the source files and other files. The compiler module 24 may be responsible for validating the directory structures and contents of various files, for example, configuration files, before proceeding with the software build process. The compiler module 24 may determine a compilation order for source code and compile each source in the identified compilation order. In some embodiments the compilation order may be defined in a file, such as a makefile, rather than determined by the compiler module 24 at the moment of starting a compile. A makefile comprises a series of structures that define build targets, compilation dependencies, the commands to build the targets, and sometimes contains references to other makefiles. A makefile is used by the make utility which may be invoked by the compiler module 24 when compiling.

[0028] Several off-the-shelf compiler programs may be invoked if the source code is in more than one programming language, because off-the-shelf compiler programs are language specific. The compiler programs typically produce an intermediate object file from each source code file. Java compilers, however, typically compile Java source code files into byte code files which are ready to run on a Java virtual machine interpreter

without any further processing. The object files and byte code files produced by the off-the-shelf compiler programs are stored in the CCS 12.

[0029] The next phase of the software build process involves a linker module 26 linking object files into executable files. The linker module 26 may comprise one or more scripts which invoke an off-the-shelf linker program. The linker module 26 has access to the object files via the CCS 12. The off-the-shelf linker program produces executable files from the object files. In some embodiments there may be no object files produced, for example, in a 100% Java based source code system, and hence in this embodiment there would be no need for a linker module 26.

[0030] At the completion of the software build process the software products are stored in the CCS 12 and are associated with a version identification. In some embodiments, an email providing notification of the result of the software build and referencing the change report 22 which controlled the software build may be sent out to members of the software development community and members of the software build administrator team. In some embodiments, the software build process may rebuild all software products. In other embodiments, the software build process may rebuild only those software products which have been affected by the changes described in the change report 22. The partial software build may be referred to as an incremental build. In an embodiment which supports incremental builds, however, a mechanism may be supported to force a complete rebuild.

[0031] The parser module 20, the compiler module 24, and the linker module 26 may not exist as separate and distinct modules or computer programs, but may be contiguous blocks of the build script. For example, build script lines 25 through 57 may be dedicated

to the responsibilities of the parser module 20, parsing comments and generating the change report 22. Build script lines 62 through 68 may be dedicated to the responsibilities of the compiler module 24. Build script lines 87 through 89 may be dedicated to the responsibilities of the linker module 26. In other embodiments, however, the build script may call other scripts, computer programs, or subroutines which correspond to the parser module 20, the compile module 24, and the linker module 26.

[0032] The parser module 20, compiler module 24, linker module 26, CCS 12, user web client 16, and administrator web client 18 include scripts and/or computer programs which may execute on a general purpose computer system, which is discussed hereinafter.

[0033] Turning now to Figure 2, a block diagram of another embodiment of a software build tool system 27 is depicted. A user interface 28 allows users to check-out and check-in code files and other files from the CCS 12. In this embodiment, the CCS 12 is a Merant PVCS version management system, but in other embodiments a different off-the-shelf CCS 12 may be employed. A code compiler module 30 comprises one or more off-the-shelf compiler programs. A document type definition (DTD) compiler 32 compiles DTD documents into metadata classes. In the preferred embodiment, the metadata classes support Vitria's BusinessWare off-the-shelf software. An interface definition language (IDL) grinder 34 transforms IDL files into code stubs. The grinder 34 transforms IDL files into Java code stubs. Grinding is the interim act of compilation that is necessary to generate dependent class files for the host application to import required data sets(s). The subsequent importing and registering of these data sets with the host application, ensures a common interface definition persists between applications/technologies. IDL stub compilation is based upon the language, for example Java or C++, and creates a class file

which then becomes the basis for subsequent code, such as Java or C++, to be compiled against.

[0034] The software build tool system 27 includes the user web client 16 interface and the administrator web client 18 interface. These web clients are supported by a web site application constructed as a standard Java Servlet/Java Server Pages (JSP) 2.3 web application. This web application is based on the Jakarta Struts web application framework. Struts is a simple web framework that provides a clean organization of the parts of a Servlet/JSP website.

[0035] The user logs on to the software build tool system 27 and triggers the parser 20 to generate the change report 22. In some embodiments, the Java properties format may be used to parse comments. The change report 22 may contain entries for multiple changed files, one entry per changed file. The user may review the change report 22 and submit the change report 22 for evaluation by the administrator. In this embodiment, the change report 22 is a Microsoft Excel spreadsheet file, but in other embodiments the change report 22 may be stored in a different file format. The software build tool system 27 may send an email containing a copy of the change report 22 in hypertext markup language format and containing a copy of a testing results document to the administrators.

[0036] The administrator logs on to the software build tool system 27 via the administrator web client interface 18 and reviews the change report 22. If the administrator decides that the software build should be launched, the administrator approves the change report 22 which triggers the software build to begin. In this embodiment, the administrator approves the change report 22 by changing the status of the change report 22 from the

CHANGED status to APPROVED status, also referred to as promoting from CHANGED status to APPROVED status, in the Merant PVCS version management system, CCS 12.

[0037] The software build tool system 27 executes a series of scripts when the software build process is started. The software build tool system 27 checks that all necessary directories exist in the CCS 12. The software build tool system 27 attempts to lock a special file in the CCS 12. The purpose of locking this special file is to assure that no other software build is being conducted at the same time, which would result in needless duplication of effort. If a lock cannot be obtained on the special file, the software build tool system 27 discontinues the software build process and sends an email to administrators stating that the special file is locked and should be unlocked. In some embodiments, the special file may be empty and plays no role other than to assure that only one build activity is occurring at a time.

[0038] The software build tool system 27 invokes the DTD compiler 32 to generate the metadata classes. The software build tool system 27 invokes the IDL grinder 34 to transform IDL files into IDL code stubs. The software build tool system 27 identifies an IDL file compilation order, a sequence for compiling the IDL files which is determined by dependencies among the IDL files, and then invokes the code compiler 30 to compile the IDL code stubs in the identified compilation order.

[0039] The software build tool system 27 invokes the code compiler module 30 to compile the source code files. In the illustrated embodiment the code compiler module 30 may determine a compilation order for source code and compile each source in the identified compilation order. In other embodiments, the compilation order may be defined in a file, such as a makefile, rather than determined by the code compiler module 30 at the

moment of starting the compile. The software build tool system 27 invokes the linker module 26 to link object files into executable files, if necessary.

[0040] The various products, including Workflow Broker build JARs, of the software build process are stored in the CSS 12 as they are completed. When the entire software build process has completed successfully version information is applied to the end products of the software build process. In some embodiments, names other than SUBMIT, CHANGED, and APPROVED may be used to denote the status of change reports 22.

[0041] The CCS 12, user web client 16, administrator web client 18, parser module 20, code compiler module 30, DTD compiler module 32, IDL grinder module 34, and linker module 26 comprise scripts and/or computer programs. Because much of the build process is controlled by scripts rather than compilable computer programs, the identification of the parser module 20, code compiler module 30, DTD compiler module 32, IDL grinder module 34, and linker module 26 may be on the basis of the function of a block of script commands rather than separate files, components, or modules which may execute as distinctly different subroutines, processes, or tasks.

[0042] In the preferred embodiment, the build scripts are Jakarta ANT scripts, some of which are extended using mechanisms provided by Jakarta ANT for extending its capabilities. To maximize the reusability of the build scripts, source and file directories may be structured to conform with current Java community practices known to those skilled in the art.

[0043] Turning now to Figure 3, a flow chart illustrating a process for building software employing the software build tool system 27 is shown. The process begins at block 50 and proceeds to block 52 where changed code is checked into the CCS 12 with a comment

conforming to a standardized comment form. This standardized form includes fields for information that is needed to build the associated software. The process proceeds to block 54 where a change report 22 is generated based on parsing the comment associated with the checked-in code provided in block 52. The process proceeds to block 56 in which an email is generated and sent to all build administrators to notify them that a change report 22 is pending authorization. The process proceeds to block 58 in which an administrator either approves or disapproves the change report 22.

[0044] At block 60, if the change report 22 is approved the process proceeds to block 62 in which the build of software is performed. The process proceeds to block 64 in which an email is generated and sent to developers and build administrators describing the build which was just completed. The process proceeds to block 66 where the process exits. If, at block 60, the change report 22 is disapproved the process proceeds directly to block 66 where the process exits without performing a build.

[0045] Turning now to Figure 4, a flow chart illustrating the detailed steps involved in building the software in block 62 above are represented. The process begins at 62a. At block 62b, if all necessary directories exist and a lock can be placed on the special file, the process proceeds to block 62c in which the DTDs are compiled into metadata classes and these metadata classes are imported into a local BusinessWare instance. Recall that the special file may be empty and plays no role other than to assure that only one build activity is occurring at a time.

[0046] The process proceeds to block 62d where those IDLs which have changed since the last build are ground into code stubs (the IDLs which remain unchanged need not be ground again, since their code stubs are still in the CCS 12). The timestamp of the last

build is kept in a file stored in each CCS 12 PVCS project build directory, for example in a file named cmac.control. Note that the DTDs are compiled before grinding the IDLs into code stubs, because some of the IDLs depend upon metadata in the DTDs. The process proceeds to block 62e where the proper compilation order for the IDL code stubs is determined and the IDL code stubs are compiled. The IDL stubs are kept in a separate directory from the other source code so that they may be zipped-up if a developer needs only the IDL stubs. Zipping a set of files is a process of packaging the files into a single file using a zip command. The files are later extracted from the zipped file using an unzip command. It will be appreciated that other compression systems may be used as well.

[0047] The process proceeds to block 62f in which code sources are compiled. Only source files which have changed are compiled, assuring a short build time. Note that the IDL stubs are compiled before the source is compiled because the source code may depend upon the IDL stubs. Where BusinessWare is used, the BusinessWare Connector def classes are called to create the customer connector INIS. INIS may refer to any structure where all the property files are stored, such that these property files or configuration files provide environment specific information based upon where they are deployed. The classes to call are stored in the project configuration file. The build is associated with a timestamp, and the lock on the special file is released.

[0048] The process proceeds to block 62g where the software build completes. If a lock cannot be placed on the special file in block 62b the process proceeds directly to block 62g and completes without having built the new software. If the build process produces object code which needs to be linked, there would be a block between block 62f and block 62g in which the object code was linked to produce an executable file.

[0049] While in the preferred embodiment the software build tool system 27 executes on a centralized build server, in other embodiments the software build tool system may execute on workstation computers dedicated to a software build administrator.

[0050] The software build tool systems 10 and 27 illustrated in the embodiments described above may complete software builds faster than less automated build systems. When making builds manually, occasionally a needed file is overlooked and not built into the new software load. This error may not be discovered until the software load is installed and executed, wasting time and delaying development progress. The software build tool system 10, 27 is readily redeployed to support a different project or the next revision stage of a project. Further, the file lock mechanism contemplated for some embodiments of the software build tool system 10, 27 avoids duplication of administrator effort.

[0051] The software build tool systems 10 and 27 also enable software builds to be run more frequently, because the build administrators's involvement is reduced and hence eliminates them as a bottle neck, which makes the software development process smoother and more efficient.

[0052] The embodiments of software build tool systems 10 and 27 described above may be implemented on any general-purpose computer with sufficient processing power, memory resources, and network throughput capability to handle the necessary workload placed upon it. Figure 5 illustrates a typical, general-purpose computer system suitable for implementing one or more embodiments disclosed herein. The computer system 380 includes a processor 382 (which may be referred to as a central processor unit or CPU) that is in communication with memory devices including secondary storage 384, read only memory (ROM) 386, random access memory (RAM) 388, input/output (I/O) 390 devices,

and network connectivity devices 392. The processor may be implemented as one or more CPU chips.

[0053] The secondary storage 384 is typically comprised of one or more disk drives or tape drives and is used for non-volatile storage of data and as an over-flow data storage device if RAM 388 is not large enough to hold all working data. Secondary storage 384 may be used to store programs which are loaded into RAM 388 when such programs are selected for execution. The ROM 386 is used to store instructions and perhaps data which are read during program execution. ROM 386 is a non-volatile memory device which typically has a small memory capacity relative to the larger memory capacity of secondary storage. The RAM 388 is used to store volatile data and perhaps to store instructions. Access to both ROM 386 and RAM 388 is typically faster than to secondary storage 384.

[0054] I/O 390 devices may include printers, video monitors, keyboards, mice, track balls, voice recognizers, card readers, paper tape readers, or other well-known input devices. The network connectivity devices 392 may take the form of modems, modem banks, ethernet cards, token ring cards, fiber distributed data interface (FDDI) cards, and other well-known network devices. These network connectivity 392 devices may enable the processor 382 to communicate with an Internet or one or more intranets. With such a network connection, it is contemplated that the processor 382 might receive information from the network, or might output information to the network in the course of performing the above-described method steps. Such information, which is often represented as a sequence of instructions to be executed using processor 382, may be received from and outputted to the network, for example, in the form of a computer data signal embodied in a carrier wave.

[0055] The processor 382 executes instructions, codes, computer programs, scripts which it accesses from hard disk, floppy disk, optical disk (these various disk based systems may all be considered secondary storage 384), ROM 386, RAM 388, or the network connectivity devices 392.

[0056] While several embodiments have been provided in the present disclosure, it should be understood that the disclosed systems and methods may be embodied in many other specific forms without departing from the spirit or scope of the present disclosure. The present examples are to be considered as illustrative and not restrictive, and the intention is not to be limited to the details given herein, but may be modified within the scope of the appended claims along with their full scope of equivalents. For example, the various elements or components may be combined or integrated in another system or certain features may be omitted, or not implemented.

[0057] Also, techniques, systems, subsystems and methods described and illustrated in the various embodiments as discreet or separate may be combined or integrated with other systems, modules, techniques, or methods without departing from the scope of the present disclosure. Other items shown as directly coupled or communicating with each other may be coupled through some interface or device, such that the items may no longer be considered directly coupled to each but may still be indirectly coupled and in communication with one another. Other examples of changes, substitutions, and alterations are ascertainable by one skilled in the art and could be made without departing from the spirit and scope disclosed herein.